Pragmatic Performance Portability with OpenMP 4.x

Matt Martineau[1], James Price[1], Simon McIntosh-Smith[1], and Wayne Gaudin[2]

[1] University of Bristol, Merchant Venturers Building, Bristol, UK
m.martineau@bristol.ac.uk,
[2] UK Atomic Weapons Establishment, Aldermaston, UK

Abstract. In this paper we investigate the current compiler technologies supporting OpenMP 4.x features targeting a range of devices, in particular, the Cray compiler 8.5.0 targeting an Intel Xeon Broadwell and NVIDIA K20x, IBM's OpenMP 4.5 Clang branch (clang-ykt) targeting an NVIDIA K20x, the Intel compiler 16 targeting an Intel Xeon Phi Knights Landing, and GCC 6.1 targeting an AMD APU. We outline the mechanisms that they use to map the OpenMP model onto their target architectures, and conduct performance testing with a number of representative data parallel kernels. Following this we present a discussion about the current state of play in terms of performance portability and propose some straightforward guidelines for writing performance portable code, derived from our observations. At the time of writing, developers will likely have to rely on the pre-processor for certain kernels to achieve functional portability, but we expect that future homogenisation of required directives between compilers and architectures is feasible.

Keywords: OpenMP 4.x, Performance Portability, Parallel Programming

1 Introduction

Today's supercomputing facilities are becoming increasingly diverse, with many hosting heterogeneous devices containing increasing levels of parallelism at the core and vector levels. As large HPC centres often need to support monolithic codes, the expense of porting codes for each new architecture is prohibitive, and given the current rate of architectural innovation, this is becoming a significant barrier to scientific progress. In order to exploit the computing resources available today, application developers have had to embrace heterogeneity and begin considering the portability of their codes [5]. The diversity of requirements presented by individual organisations means that it is going to be impossible to create a unified or one-size-fits-all solution to the current performance portability problem, but a pragmatic and forward thinking approach will go someway to protecting future HPC investment.

The OpenMP standard is a popular, mature directive-based model for targeting CPUs and, more recently, heterogeneous devices. Faced with the plethora

of parallel programming models currently available, we expect many developers will see OpenMP 4.x as a familiar and attractive option that can balance performance, portability, productivity and maintainability [8]. Of course, there are no guarantees of performance portability offered by the specification and the divergence of existing implementations means that it is currently possible to write code that is non-portable between different implementations even targeting the same architecture.

1.1 Scope

In this paper we aim to develop some best practices for performance portability by considering the different approaches taken by existing compiler vendors. We collect performance results across a range of modern devices, including those seen in large supercomputing clusters, using a suite of optimised kernels, several of which represent the performance critical functions of a range of HPC applications. The compilers that we are discussing contain bugs and lack certain features, for instance neither Clang nor GCC 6.1 provide a reduction implementation. We expect such issues to be fixed in the short to medium term, and so do not discuss these matters in any detail, and work around them wherever possible. The principal focus of this paper is on the specific design decisions made in each implementation, and how they expose long term performance portability concerns. Although we cannot guarantee complete coverage, we expect that our investigation is diverse enough that many of our insights will be applicable to general development with OpenMP 4.x.

2 Background

In July 2013, version 4.0 of the OpenMP specification was released, including a number of new directives that support targeting accelerators using computational offloading. However, up until recently the only commercially supported compiler was provided by Intel for targeting their Xeon Phi Knights Corner architecture. Some experimental compilers were developed in the interim, with the most notable being the Clang OpenMP 4.5 project, which was contributed to by a number of collaborators, including AMD, IBM, Intel, and NVIDIA. In particular, the GPU targeting functionality was developed by IBM, who are actively migrating this functionality into the main trunk of Clang [2]. In September 2015, the Cray Compiling Environment version 8.4 introduced the first official vendor support for OpenMP 4.0 on NVIDIA GPUs, with full support for version 4.0 of the specification. In April 2016, GCC 6.1 introduced support for OpenMP 4.5 offloading to HSA capable GPUs.

Readers who require introduction to the new features in OpenMP 4.x can refer to the existing literature [8, 4], and the OpenMP 4.5 specification [11].

3 Implementation-Specific Interpretations

Although the specification is very explicit about how compilers should implement the `teams` and `distribute` directives, there is some flexibility as to how the final scheduling of iterations to threads within a team is conducted. In addition to the opportunities for interpretation exposed within the specification, there are a great many implementation defined aspects of the OpenMP standard. This means that the finer details can be optimised on a per-architecture basis, making it easier for individual compilers to achieve good performance, but allowing for inconsistencies that might harm performance portability.

There is some debate regarding the prescriptive nature of OpenMP 4.x compared to the descriptive capabilities available in OpenACC with the `kernels` directive [6]. We believe that the distinction between the two approaches is actually quite small in practice, perhaps affecting the number of required directives for particular kernels. In those cases where compiler heuristic analysis of loop-level parallelism is possible and more descriptive schemes are applicable, it is not possible to guarantee the reproducibility of the parallelisation. With OpenMP 4.x, the developer certainly has to prescribe the presence of parallelism in a loop nest and direct the compiler to some extent. However, when given the minimal set of directives, the compiler has a suitable level of control over the thread co-ordination and scheduling, and how this maps to the target architecture.

3.1 Thread Co-ordination

It is useful to consider the way that each implementation maps the OpenMP model of leagues of teams of threads that can execute SIMD instructions, onto a target architecture's model, such as the CUDA model of a grid of thread blocks containing threads.

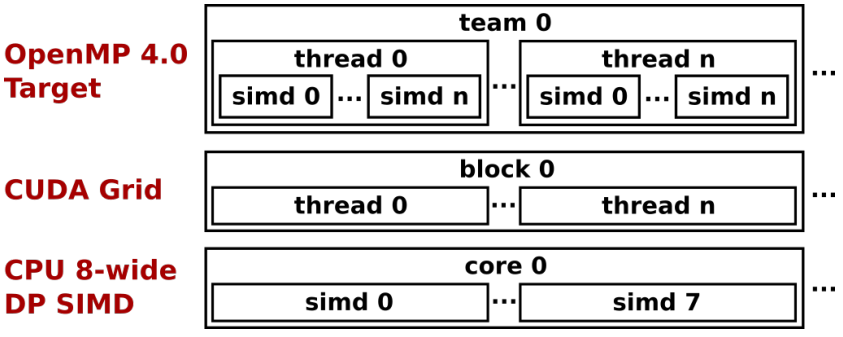

Fig. 1. OpenMP 4.x model alongside simplification of an NVIDIA GPU and Intel CPU.

Figure 1 presents a highly simplified perspective of the levels of parallelism exposed by two key target architectures alongside the OpenMP 4.x model. Please

note that the CUDA grid does not explicitly include the warps that the threads are blocked into, because CUDA implicitly handles warps and so it is not necessary to prescribe the parallelism at that granularity. There are significant overlaps between the models, but there exist subtle differences in the way that each architectural level must be handled. For instance, the CPU SIMD lanes need to be utilised in a different way to the CUDA threads, requiring the use of vector hardware instructions. In spite of this, the OpenMP model is specified such that the Cray compiler maps the teams onto CUDA thread blocks and then treats them as large vector units.

3.2 Cray Compiler Mapping of OpenMP onto NVIDIA GPUs

Each implementation adopts a different approach to mapping the OpenMP model onto their target architecture, in particular the scheduling of the iteration space across the available resources. For a simple one-dimensional loop prepended with `#pragma omp target teams distribute parallel for simd`, we have made the following observations regarding the way in which the Cray compiler mapped our OpenMP 4.5 code onto an NVIDIA K20x:

- The `teams` directive either initialises $t =$ `num_teams` teams if a value is provided, or $t = 128$ by default. The teams intuitively map to individual CUDA blocks (Figure 1), with each containing 128 CUDA threads.
- We assume that the number of OpenMP threads directly maps to the number of CUDA threads, but we were not able to prove this hypothesis given that the omp_get_num_threads() API call always returns 0.
- The `distribute` directive partitions the loop into t chunks, and distributes a chunk to the master thread of each team.
- Auto-vectorisation, or vectorisation directed by the `simd` directive, schedules the iterations in each chunk in a round robin schedule across the threads in a team, potentially wrapping such that there are multiple iterations per CUDA thread whilst maintaining coalesced memory accesses.

Code Sample 1.1. Two-dimensional kernel with outer loop parallelism.

```
#pragma omp target teams distribute parallel for
for(int ii = 0; ii < y; ++ii)
{
  #pragma omp simd
  for(int jj = 0; jj < x; ++ii)
  ...
}
```

Although we include it in Code Sample 1.1, note that the Cray compiler implementation does not explicitly require the `parallel for` directive, providing a warning upon compilation that `parallel` regions nested inside `target` regions are limited to a single thread. This warning does not mean that acceleration has failed, but that the compiler does not use the directive to guide parallelisation. In

the 2d loop case, as seen in Code Sample 1.1, the number of teams is now determined by the length y of the outer loop, such that $t = l$, and each outer iteration is associated with an independent team. While we could not determine the number of OpenMP threads instantiated, as with the 1d case, we again observed that 128 CUDA threads are created regardless of the number of OpenMP threads. The implication of this is that multiple iterations can be scheduled per CUDA thread, and in the event that fewer than 128 inner iterations are available, some of the warps will be under-utilised. The behaviour seen with two-dimensional case holds for higher dimensional loops, and collapsing can be used to revert higher-dimensional loops to the one-dimensional scheduling process.

3.3 Clang Mapping of OpenMP onto NVIDIA GPUs

Our experimentation has shown that scheduling with the Clang OpenMP 4.5 implementation uses a significantly different mechanism than that used by the Cray compiler. The compiler maps one CUDA block per multiprocessor, so when targeting an NVIDIA K20x that has 14 multiprocessors, the default is to create $t = 14$ teams. By default, each of those blocks will contain 1024 CUDA threads. When a `distribute` is encountered, the outer iterations are chunked according to the `dist_schedule`, which evenly splits the iteration space into t chunks by default. Similarly to the Cray compiler, it might be necessary for threads to execute multiple iterations.

Code Sample 1.2. Teams across the outer loop and parallel threads for inner.

```
#pragma omp target teams distribute
for(int ii = 0; ii < y; ++ii)
{
  #pragma omp parallel for schedule(static, 1)
  for(int jj = 0; jj < x; ++ii)
  ...
}
```

Clang considers the `parallel for` directive as instructing the runtime to schedule chunked loop iterations for execution by the threads in a team. This directly follows version 4.0 of the specification, which explicitly states that only when a `parallel for` region is encountered will the other threads within a team begin execution. In version 4.5 of the specification this statement has been removed, and we were not able to find a direct replacement, although the specification states that the `distribute parallel for` composite construct specifies that a loop will be executed by multiple threads of the active teams [11].

As such, to achieve reasonable performance where the outer loop is short, the `parallel for` directive must be placed on a larger inner loop, as in Code Sample 1.2, or the loops must be collapsed. It is important to recognise that the compiler does not automatically schedule iterations in a round robin order, and so when the number of iterations distributed to a team exceeds the number of threads, the directive `schedule(static, 1)` proves essential in order to enable coalescence. Please note that while we would expect kernels targeting the GPU

to use a static schedule with a chunk size of 1, this is likely not the best choice when targeting the CPU.

3.4 GCC 6.1 Mapping of OpenMP onto AMD GPUs with HSA

The GCC 6.1 implementation using HSA is currently restricted to a single combined construct `target teams distribute parallel for`. This limitation is strict, and clauses such as `collapse` are not implemented when targeting HSA enabled devices. Although we were not able to use the OpenMP API calls: `omp_get_thread_num()` etc, we analysed the source code in order to ascertain the mapping scheme. Unsurprisingly, this implementation took a different approach to both the Cray compiler and Clang, mapping OpenMP teams as work groups containing 64 work items. The number of work groups, or teams, launched is the size of the iteration space n, divided by the number of threads in a single team t.

3.5 Intel Mapping of OpenMP

Although Intel's offloading capability was primarily targeted towards the Intel Xeon Phi Knights Corner architecture, it is still useful to understand their design decisions from the perspective of performance portability. In spite of the `teams` directive, the Intel compiler only initialises a single team by default, and as such both the `teams` and `distribute` directives can be omitted, although we do not advise this for performance portability. Essentially, the compiler offloads the loops using the CPU approach of threading over the outer loop and vectorising an inner loop, or performing both on the outer loop if one-dimensional or collapsed.

4 Performance Analysis

As many of the implementations are new or experimental and had some deficiencies, it was not possible to collect results across all of the compilers and devices using full applications. Instead we have chosen representative kernels, including several that are performance critical within HPC applications. While we don't explain every kernel in detail, the names serve to describe the basic function, and the source code can be found in our open source repository[3].

The results in Figure 2 represent the performance data collected for this research in full, and have been sampled across multiple architectures: an NVIDIA K20x GPU, a 44 core Intel Xeon Broadwell CPU, both resident in the Cray XC40 Swan supercomputer, as well as an Intel Xeon Phi Knights Landing (KNL) 7210 and AMD A10-7850K Radeon R7 (Kaveri) APU hosted at the University of Bristol. The CUDA application serves to demonstrate the performance achieved with a naive parallelisation of each kernel on a K20x, collapsing the iteration space of all kernels into a one-dimensional grid containing blocks of 128 threads.

[3] https://github.com/UoB-HPC/pragmatic_kernels

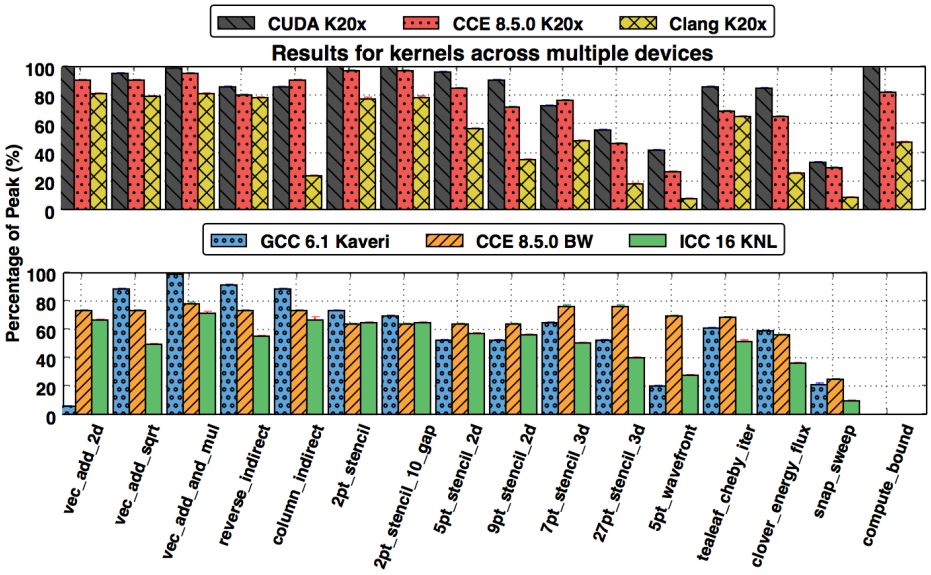

Fig. 2. Kernel performance *(higher is better)*: CCE, Clang, CUDA on K20x (182 GB/s), CCE on 44 Core Broadwell (125 GB/s), ICC on KNL 7210 (439 GB/s), and GCC 6.1 on AMD Kaveri APU (5.4 GB/s).

The Clang results specifically use the `clang-ykt` implementation, which is no longer supported, but represents the most functional open-source Clang implementation that can target GPUs with OpenMP 4.5. GCC 6.1 has a highly constrained interface, providing only a single combined construct, which completely limits the ability to perform optimisation. The results for all kernels, except for `compute_bound`, represent the proportion of peak memory bandwidth.

4.1 Individual Performance

The `vec_add*` kernels serve as a simple baseline for performance, and we expect them to achieve a good proportion of peak. In most cases this proves to be correct, and the results are quite consistent, however the KNL suffers from a noticeable reduction in performance for the `vec_add_sqrt` kernel. As a side note, we observed that the performance on the KNL varied more than expected as the problem size is altered, and achieving peak performance for most of the kernels required the working set to approach the total memory capacity of the device. The Cray compiler is within around 10% peak bandwidth of the CUDA kernels, while Clang achieves within around 20% bandwidth, which likely demonstrates the latent overheads present in both implementations. The `vec_add_2d` kernel exposes a performance issue with GCC 6.1, as the nested loops mean that the performance is 20x lower than expected. As the `collapse` statement is not yet supported, loops currently require manual collapsing to achieve reasonable per-

formance. For all other kernels we have manually collapsed the loops to observe some meaningful results.

The `*indirect` kernels use indirection arrays in their loops, which is a pattern that we have isolated as challenging to accelerate in other applications. The `reverse_indirect` kernel is an example where the Cray compiler achieved a 3.5x speedup through using the `collapse(n)` statement. Clang has poor performance for the `column_indirect` kernel, and this is because the inner loop cannot be collapsed into the iteration space, which limits the available work to the length of the outer loop. The `column_indirect` kernel is an instance where our OpenMP 4.0 implementation with the Cray compiler has beaten our CUDA implementation, by virtue of the simple scheduling we have performed with all of the CUDA kernels. In our OpenMP 4.x implementations we have only collapsed the loops where the performance was improved, and in this case performance was better if the inner loop was partitioned rather than collapsed into the iteration space. While the results on the Broadwell and using GCC 6.1 are consistent for these kernels, the KNL has slightly worse performance for the `reverse_indirect` kernel, although we were not yet able to ascertain the cause.

For the stencil operations, the achieved memory bandwidth generally decreases as the size of the stencil increases from 2 up to 27. The Cray compiler stays within 20% of the peak bandwidth compared to CUDA for all of the kernels, and slightly improves upon the naive CUDA scheduling strategy for the 7pt stencil. The Clang compiler achieves reasonable performance for the small stencils but the 9pt and 27pt stencils drop below 50% bandwidth compared to CUDA. We did consider that this may be a byproduct of the potential for increased register pressure associated with the larger stencils, but upon checking we discovered that only 48 registers were utilised and the occupancy was above 50% for both of those kernels, suggesting this is not the issue. The performance of the other implementations was fairly consistent across the kernels.

Our `five_point_wavefront` kernel represents a unique and challenging parallel data traversal. The Cray compiler attains tolerable performance of around 60% of the bandwidth compared to CUDA, but Clang achieved under 20% of CUDA's in spite of all efforts to optimise the kernel. The Broadwell performs well with this kernel, while the KNL results are quite low, but we note that increasing the problem size did improve the performance up until device capacity. Even after manually collapsing the kernel's loop nest, the GCC 6.1 implementation does not perform particularly well with this kernel either.

The application kernels from TeaLeaf, CloverLeaf, and SNAP are important because they provide an indication of the performance that might be seen in a production scientific application. All of the compilers achieve good performance for the `tealeaf_cheby_iter` kernel on all devices. We note that the TeaLeaf kernel is a case where using the `collapse(n)` clause indiscriminately leads to poor performance, reducing the kernel's peak bandwidth on a K20x compiled with the Cray compiler from 117 GB/s to 66 GB/s. The loopmark listing file provided by the Cray compiler states 'rediscovery of loop control variables' is introduced, which might be causing an overhead contributing to the reduction in

performance, but is likely the sole cause of such a large performance decrease. It is not clear why the performance is not satisfactory, and the only difference between this kernel and others in the suite is the extensive use of ternary conditionals and `fabs`.

We observed that utilisation of peak bandwidth was poor for the `snap_sweep` kernel across all devices, and find similar performance to the 5pt wavefront algorithm, as would be expected given their similarities. On the K20x the Cray compiler is within 10% of the performance of the CUDA implementation, while the Clang implementation achieves only 24% of CUDA's peak. All of the other implementations achieve low memory bandwidth, but this is no fault with the implementations as the SNAP kernel is not memory bandwidth bound.

The `compute_bound` kernel executes 128 statements that can be transformed into fused-multiply-add (FMA) instructions, to demonstrate that there is a disparity in the usage of FMAs between the NVIDIA GPU targeting compiler implementations. By inspecting the generated PTX we were able to confirm that the Clang compiler does not transform the statements into FMAs, whereas the Cray compiler does.

4.2 Directives for Performance

Often, the key to achieving good performance was collapsing loops, and placing the `simd` in the appropriate place to enable vectorisation. For the `clang-ykt` version of Clang that we tested, adding `schedule(static, 1)` was also essential, but we have been informed that this will not be a requirement when OpenMP 4.5 functionality is feature complete in the Clang trunk. Kernels like `compute_bound` and `snap_sweep` required the use of `simd`, but the placement of this particular directive was somewhat dependent upon the architecture, and some implementations would suffer a significant performance hit when adding `simd`, as it can change the parallelisation from an optimal scheme.

The OpenMP 4.5 specification stipulates that all scalar variables will default to `firstprivate`, whereas version 4.0 implicitly maps scalar variables at the beginning and end of a kernel. This original behaviour means that there is a small overhead caused by the copying of scalar variables around target invocations. The OpenMP 4.0 workaround is to declare an explicit mapping using `map(to: scalar variables)`. This will make little difference for kernels with lots of work within a single parallel region, but our sweep implementation required many short kernels to be executed across each of the planes. Even though the individual copies took only μs, this happened twice at the end of each plane within the spatial domain, and by mapping the scalar variables as `to` only, we observed a noticeable improvement in runtime. Importantly, for compilers that do not implement the OpenMP 4.5 default data sharing rule, this optimisation is effective and does not hinder performance portability.

In general, we found that achieving performance for all of the devices with all of the compilers was not necessarily trivial. However, even when using preprocessor macros to include compiler-specific directives at the loop level, the benefits compared to managing multiple lower-level codes cannot be overstated. It is clear

from the results that we were able to achieve a good level of performance across a range of devices using a single intuitive programming model.

5 Approaching Pragmatic Performance Portability

At the time of writing, it is valid and correct to write OpenMP 4.x code that targets CPU, GPU and KNC using significantly different sets of directives (Code Sample 1.3).

Code Sample 1.3. Different approaches to loop level parallelism.

```
// (a) Example directives for Cray targeting NVIDIA GPUs
#pragma omp target teams distribute simd
for(...)

// (b) Example directives for Clang targeting NVIDIA GPUs
#pragma omp target teams distribute parallel for schedule(static, 1)
for(...)

// (c) Example directives for GCC 6.1 targeting AMD GPUs
#pragma omp target teams distribute parallel for
for(...)

// (d) Example directives for Intel targeting KNC and CPU
#pragma omp target if(offload)
#pragma omp parallel for
for(...)
```

Unfortunately, this divergence in accepted directives means that there is the potential for functional portability issues between the different compilers. All of the options are valid for the Intel compiler, the Cray compiler will also accept (b) and (c), and Clang will accept (c), but will likely perform poorly. GCC is the most constrained, and will only work with the exact construct listed in (c). This is a small but important example of the potential pitfalls that a developer can encounter when developing OpenMP 4.x applications, and it is possible that future implementations from other vendors will make the situation more complicated. Observe that (d) uses the `if` conditional clause to disable the `target` if the CPU is being targeted. This functionality can alleviate some portability issues, allowing the same kernels to be conditionally run on the host or offloaded. Version 4.5 of the OpenMP specification extends the conditional clause to allow the form `if(directive: condition)`, such that both the `target` and `parallel` directives can be conditionally disabled.

It might be possible to extend this conditional functionality to switch on and off the different directives based on the target type, however this suffers from the same issues as using preprocessor conditions, and may end up harming the potential for long term portability. It would be preferable for the developer to be able to express what parallelism exists at the loop-level and then allow

the compiler to choose which levels are applicable to the particular target. For instance, on the CPU we might only be concerned with partitioning an outer loop across cores and executing an inner loop with vector instructions. The `parallel` loop construct and `simd` directives are purpose-built to achieve this partitioning, but this is not the only option available with the new directives introduced in version 4.5. The same scheme can be described using the `teams distribute` directives, by limiting each team to a single thread on the CPU, allowing the `simd` directive to describe vectorisation of the inner loop.

We believe that, in order to improve the potential for future functional portability, developers need to aim to provide the most encompassing description of loop-level parallelism possible. Whilst giving as much information as possible is effective, a balance must be struck to avoid inhibiting the ability for the compiler to automatically optimise the scheduling and tuneable widths for each architecture. Essentially, this entails using as many of the *general* directives as possible, as seen in Code Sample 1.1. The natural approach is to use the combined construct `target teams distribute parallel for` to describe the parallelism available at the team and thread level, and the `simd` statement to direct vector level parallelism.

5.1 Homogenising the Directives

Reducing the standard set of directives into an encompassing group was not entirely possible, but we did make progress. In particular we were able to create a set of directives for most kernels using Clang and the Cray compilers. Clang requires `schedule(static, 1)` for performance, but the Cray compiler defaults to this schedule, and so including the directive did not harm performance. A significant obstacle for performance portability was the `simd` directive, as the combined construct `target teams distribute parallel for simd` is not available with GCC 6.1 and negatively affected the performance achieved by Clang.

We did observe that the `collapse(n)` statement is essential for performance in some cases and harmful in others, which made it impossible to merge directives in many cases. Another example where homogenisation was challenging is the `column_indirect` kernel, where the `parallel for` directive had to be added to an inner loop for performance with Clang, but this made the performance unacceptable for Cray. We also noticed that it was essential for performance on the KNL that all methods vectorised successfully, and so this meant using the `simd` statement far more regularly than was necessary with the Broadwell. Overall we have found that there will need to be some progress towards standardisation for future functional inter-compiler portability, and to enable performance portability with homogenised directives.

5.2 Patterns That Can Inhibit Performance Portability

An interesting pattern demonstrated in Code Sample 1.4 uses an indirection on the inner loop that simply contains the value of x in all elements, but inhibited

the potential for collapsing. When using Clang, this meant we had to parallelise the inner loop with `parallel for`, and suffered a 3.5x increase in runtime compared to the same kernel without the indirection.

Code Sample 1.4. Indirection use on inner loop.

```
for(int ii = 0; ii < y; ++ii)
{
  for(int jj = 0; jj < indirection[ii]; ++jj)
  {
    ...
  }
}
```

Certain algorithmic patterns appear to contain dependencies that inhibit successful acceleration. The `snap_sweep` kernel is a good example of this, as it uses indirections that are accessed with variables evaluated at runtime, which often resulted in variable success when attempting to parallelise the kernel. We expect that as the implementations are improved in the future, a strict adherence to the developer's independence guarantees are provided. For instance, implementations choosing to map the scheduling of threads across warps within a team as the vectorisation of some inner loop, should always infer that the loop iterations can be executed concurrently as given by the `simd` directive, if it is provided. This will allow the developer to achieve consistent parallelisation without having to restructure the code to support the compiler.

When testing CCE 8.5.0 and Clang, we noticed that our timing code was reporting incorrect results. It transpired that each of the kernels is queued asynchronously as a task, and so our timing between the calls was incorrect. We expected that a directive such as `wait` or `taskwait` would have been well placed to perform the synchronisation that we required, but this was not possible, so we had to rely upon an unnecessary read of a scalar from the device to force synchronisation.

The `collapse` Clause has an important role in the performance portability of OpenMP 4.x applications and, depending on the application, may have a more significant semantic impact than developers would expect. The specification states that the `collapse` statement determines which loops a `distribute` directive will partition, and each loop that is collapsed will have its iteration space combined into a single space. While this is functionally identical to `collapse` relative to a `parallel` region, we have shown in Section 3.1 that the design of current implementations means that the `collapse` clause can fundamentally alter the way that thread scheduling occurs for a particular set of loops.

While it may seem tempting to add `collapse(n)` to the stock set of directives included at every loop, we reiterate that on several tests, the Cray compiler suffered a significant performance hit when collapsing loops indiscriminately. In particular, we noticed that a performance penalty will be likely when a loop nest incorporated halo padding, presumably demonstrating the overhead of the

more complicated scheduling required once the loop is collapsed. Contrary to this, the clause is essential for increasing the work available to the device. It is imperative that enough work is provided to the device but it isn't necessarily trivial to determine the effect of the `collapse` clause when considering multiple kernels, across multiple devices each with different parallelisation schemes for different devices. As such, we can only suggest that the clause is used judiciously and testing is performed for realistic problem sizes to ensure that it is actually necessary for a particular kernel.

5.3 Concluding Suggestions for Performance Portability

While it is not possible, at the time of writing, to write a single set of directives and achieve functional portability across the range of compilers and devices, we believe that homogenisation will head in a predictable direction. As such, we present some tips that might help future proof codes using OpenMP 4.x:

- *Prefer to include the most extensive combined construct relevant to the loop nest e.g.* `#pragma omp target teams distribute parallel for simd`. The combined constructs are easier to reason about, and more consistently interpreted between compiler implementations.
- *Always include* `parallel for`*, and* `teams` *and* `distribute`*, even if the compiler does not require them.* Excluding them for compilers that use exclusive mechanisms to map onto the target architecture will inhibit acceleration on other devices, and execution on CPUs.
- *Include the* `simd` *directive above the loop you require to be vectorised.* Being explicit about vectorisation improves the chances that all target compilers will succeed in accelerating the code with the intended results.
- *Neither* `collapse` *nor* `schedule` *should harm functional portability but might inhibit performance portability, so prefer not to include them when possible.* It will be essential to use `collapse(n)` for certain loop nests and compilers, but it should not be included blindly. We expect that future compiler versions targeting devices supporting coalesced memory accesses will default to using `schedule(static, 1)`, and so it might be better for future portability between those devices and the CPU to avoid the clause.
- *Avoid setting* `num_teams` *and* `thread_limit`*.* Each compiler uses a different scheme for scheduling teams to a device. Making minor adjustments to improve performance with one device might significantly reduce performance on other devices. It would be preferable to only use the clauses where there are performance critical loops that cannot perform with the compiler defaults.

Of course, there will be occasions in applications where these guidelines cannot be followed, and current compilers do not necessarily support the directives and clauses such that future-proof code will execute correctly. For instance, the branched Clang version of OpenMP 4.0 performs poorly with the `simd` directive, and GCC 6.1 targeting HSA does not support any clauses.

6 Related Work

Hart et al. [4] ported the NekBone mini-app to use the Cray compiler's OpenMP 4.0 GPU offloading functionality, detailing the porting process and subsequent optimisation. Bercea et al. [1] analysed the performance of their OpenMP 4.0 port of the CORAL proxy application, and discussed the impact of register spilling. Lin et al. [7] used the ROSE source-to-source compiler to port a number of stencil applications, investigating performance and productivity. In our previous work, we compared the performance of a number of parallel programming models, including OpenMP 4.0, Kokkos, and RAJA [8]. We later discussed the performance of OpenMP 4.0 ports of the TeaLeaf, CloverLeaf, and BUDE mini-apps on NVIDIA GPUs [9]. In some of our earlier performance portability work, we investigated the performance of OpenCL with several structured grid codes, demonstrating a number of techniques that lead to performance portability [10]. Bertolli et al. [3] discuss the coordination of threads within an NVIDIA GPU, and show that their novel approach limits the impact on code generation when integrated into the LLVM compiler infrastructure. They later discussed their approach to integrating OpenMP 4.5 offloading for NVIDIA GPUs into Clang [2].

7 Acknowledgements

We would like to thank Cray Inc. for providing access to their XC40 supercomputer Swan, which hosted the Intel Xeon Broadwell, and NVIDIA K20x processors. The Intel Xeon Phi KNL was provided by the Intel Parallel Computing Center at the University of Bristol, and we would like to thank Jim Cownie at Intel for his support. We also want to thank the sponsors of this research, EPSRC and the UK Atomic Weapons Establishment.

8 Future Work

While this research has focused purely on data-parallel applications, it will be important to consider the task-parallel capabilities of the specification. It would be useful to track the progress of each of the available compilers, as well as investigating new implementations as they become available. Further to this, the Clang compiler that we used is out of support and as soon as the newest version has been promoted to the trunk it will be important to understand the difference in the parallelisation scheme and performance, if any.

9 Conclusions

Performance portability is not guaranteed by the OpenMP 4.5 specification, and the individual compiler implementations suffer from a number of limitations. The different compiler vendors have interpreted the specification such that it is

possible for developers to write codes that are tightly coupled to a single implementation. We have found that good performance can be achieved across a range of HPC devices, using several different implementations. Having tracked the progress made within the last year, there is now strong evidence that performance portability is possible using OpenMP 4.x, and while standardisation and coherence are needed between compiler vendors, the responsibility falls on the developer to prefer portable practices.

References

1. G. Bercea, C. Bertolli, S. Antao, A. Jacob, et al. Performance Analysis of OpenMP on a GPU using a Coral Proxy Application. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, page 2. ACM, 2015.
2. C. Bertolli, S. Antao, G.-T. Bercea, et al. Integrating GPU Support for OpenMP Offloading Directives into Clang. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, 2015.
3. C. Bertolli, S. F. Antao, A. Eichenberger, et al. Coordinating GPU threads for OpenMP 4.0 in LLVM. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, pages 12–21. IEEE Press, 2014.
4. A. Hart. First Experiences Porting a Parallel Application to a Hybrid Supercomputer with OpenMP 4.0 Device Constructs. In *OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015, Proceedings*, pages 73–85, 2015.
5. P. Kogge and J. Shalf. Exascale Computing Trends: Adjusting to the "New Normal" for Computer Architecture. *Computing in Science & Engineering*, 15(6):16–26, 2013.
6. J. Larkin. Performance Portability Through Descriptive Parallelism. Presentation at DOE Centers of Execellence Performance Portability Meeting. Available from: https://asc.llnl.gov/DOE-COE-Mtg-2016/talks/2-20_Larkin.pdf, 2016.
7. P. Lin, C. Liao, D. Quinlan, et al. Experiences of Using The OpenMP Accelerator Model to Port DOE Stencil Applications. In *OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015, Proceedings*, pages 45–59, 2015.
8. M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin. An Evaluation of Emerging Many-Core Parallel Programming Models. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'16, 2016.
9. M. Martineau, S. McIntosh-Smith, and W. Gaudin. Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model. In *Proceedings of 21st International Workship on High-Level Parallel Programming Models and Supportive Environments*, HIPS'16, 2016.
10. S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price. On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures. In *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 53–75. Springer International Publishing, 2014.
11. OpenMP Architecture Review Board. OpenMP Application Program Interface v4.5, 2015.